Brian Erichsen Fagundes
Conclusion
CS6013 - MSD
UofU
Spring Semester 2024
Malloc Replacement Assignment

The observed performance discrepancy between the system allocator's and my malloc's class allocation and deallocation methods implementation arises from several key factors inherent to the simplicity of the design of how I developed my hash table and my B_Malloc's class. For context; the time taken for 100 allocations and deallocations using the system allocator average was of 0.000136417 seconds while my own implementation of allocating and deallocating the same amount of resources was of 0.00140958 seconds. This discrepancy shows that my implementation takes 10 times more on average than using the standard system allocator. The major reasons that my implementation is slower than the standard are related to page size and fragmentation, and linear probing hash table overhead.

One major contributor to the performance gap is my B_Malloc class reliance on page-sized allocations. The **mmap** system call allocates memory in page-sized chunks, leading to increased internal fragmentation. Internal fragmentation occurs when we split the physical memory into contiguous mounted-sized blocks and allocate memory for a process that can be larger than the memory requested; hence the unused allocated space is left and cannot be used by other processes. In the B_Malloc's implementation smaller allocations result in a waste of memory, as they are rounded up to the nearest page size. In contrast, the system allocator employs more sophisticated algorithms to manage memory efficiently which reduces internal fragmentation.

The linear probing hash table used in the B_Malloc's class introduces overhead during allocation and deallocation. The probing mechanism can lead to cache misses, impacting lookup and insertion times. Additionally, the hash table itself requires additional memory and management, contributing to the observed performance overhead.

How to Make My Implementation More Sophisticated, Better, and Faster?

Memory pooling: Implementing a memory pooling strategy can improve allocation performance. Instead of relying solely on mmap, a pool of pre-allocated memory blocks could be managed for specific size classes. This would help reduce the impact of internal fragmentation by providing more granularity in memory allocation. However, while memory pooling can reduce fragmentation, it may come with a cost of increased overhead due to managing multiple pools. Even with the cost of increased overhead, memory pooling benefits of reduced fragmentation and improved allocation efficiency often outweighs this cost.

Improved Hashing: Enhancing the hash function and exploring alternative hash structures might mitigate linear probing-related overhead. Considerations for open addressing techniques, such as using the quadratic probing, where it avoids table collision by probing for the next available slot using a quadratic function instead of linear logic used in my implementation. For the proposed solution of improving hashing by enhancing the hash function and exploring alternative hash structures, there are potential trade-offs to consider. One trade-off is the complexity introduced by implementing more sophisticated hash functions or alternative hash structures. While these enhancements may reduce linear probing-related overhead and improve lookup and insertion times, they can also increase the complexity of the codebase and require additional computational resources. For example, implementing a more complex hash function may require additional CPU cycles for computation, potentially impacting overall performance. In spite of increased complexity when hashing is improved the potential gains in performance can justify these trade-offs.

Thread-Specific Pools: Implementing thread-specific memory pools can reduce contention among threads, potentially improving parallelism and overall allocation speed. This approach would minimize the need for locks when managing the hash table.

In summary, while my implementation provides a functional allocator, its simplicity comes at the cost of performance. Adopting advanced memory management techniques and optimizations, as outlined above, can significantly enhance the speed and efficiency of my allocator, moving it closer to the performance level of the system version.